The Pre-images of Bitwise *AND* Functions in Forensic Analysis

Kyriacos E. Pavlou and Richard T. Snodgrass

October 10, 2006

TR-87

A TimeCenter Technical Report

Title The Pre-images of Bitwise *AND* Functions in Forensic Analysis

Copyright © 2006 Kyriacos E. Pavlou and Richard T. Snodgrass. All rights reserved.

Author(s) Kyriacos E. Pavlou and Richard T. Snodgrass

Publication History October 2006. A TIMECENTER Technical Report.

Abstract

Mechanisms now exist that detect tampering of a database, through the use of cryptographically-strong hash functions, as well as algorithms for *forensic analysis* of such tampering, thereby determining who, when, and what. This paper shows that for one such forensic analysis algorithm, the *tiled bitmap algorithm*, determining when and what, which goes far in determining who, has at its core computing the pre-image of bitwise *AND* functions. This paper introduces the notion of *candidate set* (the desired pre-image of a target binary number), provides a complete characterization of the candidate set, and characterizes the cardinality as well as the average cardinality. We then provide an optimal algorithm for computing the candidate set given a target. We show that given an auxiliary *candidate array*, the candidate set can be computed in constant time. This latter algorithm is applicable when the target is a suffix of another target number for which a candidate set has already been computed.[1]

1 Motivation

Due in part to recent federal laws (e.g., Health Insurance Portability and Accountability Act: HIPAA, Sarbanes-Oxley Act), and in part due to widespread news coverage of collusion between auditors and the companies they audit (e.g., Enron, WorldCom), which helped accelerate passage of the aforementioned laws, there has been interest in built-in mechanisms to detect or even prevent database tampering.

We previously proposed an innovative approach in which cryptographically strong one-way hash functions prevent an intruder, including an auditor or an employee or even an unknown bug within the DBMS itself, from silently corrupting the audit log [5]. This is accomplished by hashing data manipulated by transactions and periodically *validating* the audit log database to detect when it has been altered. Validation involves sending the hash value computed over all the database to an external *notarization service*, which will indicate whether that value matches one previously computed. Should tampering have occurred, the two hash values will not match.

The question then arises, what do you do when an intrusion has been detected? At that point, all you know is that at some time in the past, data somewhere in the database has been altered. *Forensic analysis* is needed to ascertain *when* the intrusion occurred, *what* data was altered, and ultimately, *who* the intruder is.

Validation provides a single bit of information: has the database been tampered with? To provide more information about when and what, during validation we hash across subsets of the database. As the database transactions that are hashed occur in commit order, each set of values that is hashed is referred to as a *hash chain*. Then, during forensic analysis of a subsequent validation that detected tampering, those chains can be rehashed to provide a sequence of truth values (1 = Success and 0 = Failure), which can be used to narrow down where and what.

We have proposed a variety of *forensic analysis algorithms*, differing in the amount of work necessary during normal processing (computing additional hash chains during periodic validation) and the precision of when and where during forensic analysis [2].

In the *tiled bitmap algorithm* [3], a variant of the *polychromatic algorithm* [2], the hash chain groups are aligned with the actual validation intervals. In Figure 1, validation occurs each 16 hours, during which time many thousands or even millions of transactions occurred; this figure shows one 16-hour slice. (Time proceeds left to right, with the hour shown as $r = 0$ to $r = 15$. This is the second slice, so hour 17 corresponds to $r = 0$ and hour 28 corresponds to $r = 11$.)

This figure illustrates a *corruption event* in which the timestamp of a tuple in a relation was changed from hour 31 to hour 28. Consider that the relation records when privacy release authorizations were signed by a patient; in this case the authorization was signed in hour 31. A doctor revealed health information to an insurance company some confidential information on hour 29, then, later realizing his mistake, which is an offense under HIPAA, backdated that authorization to hour 28, as shown by the left-pointing arrow. Thus, the database now implies that authorization had been received before the confidential information was transferred. (The fact that this backdating was done on hour 47 will be revealed by a separate part of the forensic analysis algorithm, not discussed here. The details of this algorithm may be found elsewhere [2, 3].)

[1]The authors are at the Department of Computer Science, University of Arizona, Tucson, AZ, {kpavlou,rts}@cs.arizona.edu This research was supported in part by NSF grants IIS-0415101, IIS-0639106, and EIA-0080123 and with partial support from a grant from Microsoft Corporation.

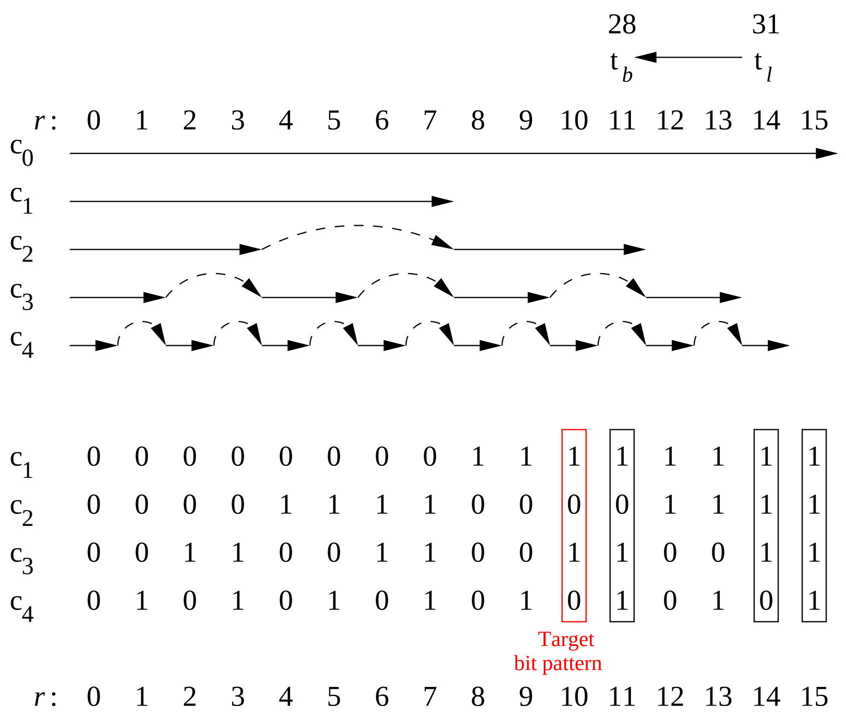
28
31
t b
t l
r: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
c0
c1
c2
c3
c4
c1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
c2 0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
c3 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1
c4 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
Target
bit pattern
r: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Figure 1: Hash Chains in the Polychromatic Forensic Analysis Algorithm

We wish for forensic analysis to constrain the corruption to within an hour, so the database administrator configures the database management system to compute five hash chains, c_0 through c_4, also shown in the figure. The first hash chain, c_0, hashes all transactions within the notarization interval of 16 hours, in order, to compute a hash value. (This hashing was done during a validation previous to the one that detected the corruption.) It is this hash value that narrows down the corruption event to this particular validation interval. The second hash chain, c_1, hashes only the first 8 hours worth of transactions. The last hash chain, c_4, hashes every other hour's worth. The dotted line indicates linking of hash chains. For example, in hash chain c_4, hash value of the last transaction of hour 0 is hashed with the hash value of the first transaction in hour 2. Hash chain linking is discussed in more detail elsewhere [5].

When the tampering is detected, sometime after hour 47, the hash value for each chain is recomputed, on the tampered data and sent to the notarization server, which responds with "success" (the old and new values match) or "failure". Hash chain c_0 reports failure, which means something that was stored during this 16-hour period was tampered with. Changing the timestamp on a tuple is equivalent to removing that tuple from all hash chains that cover the original time and adding that tuple to all hash chains that cover the inserted time. We have four remaining hash chains, so we compute a 4-bit value from this corruption event: 1010 (reading from c_1 as the high-order bit and c_4 as the low-order bit). The reason is that the hash value of chains c_1 and c_3 were not effected by the corruption, as neither of these chains include hours $r = 11$ or $r = 14$; the hash value of chains c_2 and c_4 no longer match those previously computed.

The truth values shown at the bottom of the figure indicate the target string that would result had the corruption event tampered with data stored at the indicated hour. For example, changing the data of a tuple that was originally stored in the first hour of this interval would have rendered all of the chains as failure, resulting in a value of 0000. We term this value the *target binary number*, or *target*. The target is the input to the forensic analysis.

For our corruption event that occurred at hour 47, changing a timestamp from 31 to 28, the hash chains provide a target of 1010. What could such a target indicate? One possibility is that only the data in hour 27 ($r = 10$) was modified. Another is that the timestamp was moved from 28 ($r = 11$) to 31 ($r = 14$). A third possibility is that the time was moved from 31 to 28. Other possibilities are a change from hour 27 ($r = 10$) to 31, a change from hour 32 ($r = 15$) to hour 27, or a change in the other direction. All these possibilities result in a target of 1010.

Because the "from" time and the "to" time both occur within the same validation interval and because the hash chains are linked together, then the bit patterns given above are *AND*ed, and the resulting target of 1010 corresponds to the existence of (a) either a single suspect day, for a data corruption, or (2) two suspect days, for a pre- or post-dating corruption, or (3) some combination thereof. It is important to note that corruptions on some hours could not have produced this target. For example, given that c_3 succeeded, we know that nothing was affected in hours 0, 1, 4, 5, 8, 9, 12, or 13.

In reality the situation is more complicated since when dealing with multiple corruption events there might be many combinations of bit patterns that can yield the single bit pattern the forensic analysis will produce (target bit pattern). This is also true even in the simple case where a single post/backdating corruption event *does not* have its endpoints in distinct hash chain groups. In other words, the pre-image of the target bit pattern under the *AND* function is not unique. We address this complication in the remaining of this paper. Our task is to extract from a single target all the possible corruption events, the core computation of which is determining the pre-image.

2 Problem Formulation

We define the length l of a binary number b, denoted by $|b| = l$, as the number of its digits. From this point forward we consider l to be fixed. We seek to find the pre-images of all the binary numbers of length l, $\mathbb{B} = \{b : |b| = l\}$, under a family of bitwise *AND* functions whose domain is a finite Cartesian product.

$$\mathit{AND}_k : \mathbb{B}^k \longrightarrow \mathbb{B}$$

$$\mathit{AND}_k((b_1, b_2, \ldots, b_k)) = b_1 \wedge b_2 \wedge \ldots \wedge b_k$$

Observe that the maximum number k of sets participating in the Cartesian product is 2^l, since if k is allowed to take a value beyond that, it will force a repetition of one of the binary numbers. This is not informative or useful in any way since repeated *AND*ing operations with the *same* binary number leave the result invariant (the operation is *idempotent*). In other words, repetition is not allowed and hence for a given k-tuple all its components are distinct. Also note that the value of k uniquely identifies a specific AND_k function in the above family.

We name the set of all binary numbers which appear as components in at least one of the pre-images (i.e., k-tuples) of a specific target binary number t the *candidate set*:

$$C_{t,k} = \{b \in \mathbb{B} : \exists\, b_1, b_2, \ldots, b_{k-1} \text{ s.t. } \mathit{AND}_k((b, b_1, b_2, \ldots, b_{k-1})) = t\}$$

The $\wedge$ operation is commutative: the order of the operands does not matter, and that is why this is a set. The word "candidate" was used to name this set because, in the original formulation of the problem, its elements correspond bijectively to granules (the units of time a database is partitioned into, e.g., the hours indicated in Figure 1), which are candidates where corruption may potentially be detected. For the example provided before, the candidate set would be the hours 27, 28, 31, and 32 that is, $r = 10$, $r = 11$, $r = 14$, and $r = 15$. In this forensic analysis context, the value of k represents the *actual* number of granules corrupted.

For convenience we can express these sets in decimal, though our algorithms read and write in binary. For example:

$C_{1010,1} = \{1010\} = \{10\}$

$C_{1010,2} = \{1010, 1011, 1110, 1111\} = \{10, 11, 14, 15\}$

1001 is not in $C_{1010,2}$ because 1001 cannot be in the pre-image of 1010. Note that even though two binary target strings may have the same numerical value, if their length is different then their candidate sets will be different. For example, the candidate set $C_{000,2}$ is different from $C_{0000,2}$.

We wish to characterize formally the candidate set and develop algorithms for efficiently calculating this set, given t and k.

3 Characterizing the Candidate Set

Let $z(t)$ be the number of zeros in the binary number t, e.g., $z(1010) = 2$. By definition $1 \leq k \leq 2^l$ and $0 \leq z \leq l$. The behavior of $C_{t,k}$ for increasing k is interesting. As k increases the candidate set for a fixed t remains invariant and equal to the candidate set for $k = 2$, until some threshold value $2^{z(t)}$ after which it becomes empty. Simply put, $C_{t,k}$ obeys an all-or-none law. A complete characterization of the candidate sets is given below.

Theorem 1.

$$C_{t,k} = \begin{cases} \{t\} & , \quad k = 1 & (1) \\ \emptyset & , \quad z(t) = 0 \land k > 1 & (2) \\ C_{t,2} \neq \emptyset & , \quad l \geq z(t) > 0 \land 2 \leq k \leq 2^{z(t)} & (3) \\ \emptyset & , \quad l \geq z(t) > 0 \land k > 2^{z(t)} & (4) \end{cases}$$

Proof. Case (1): $k = 1$
We want to find the binary numbers that map to t. In this case $k = 1$, i.e., the pre-image is unique and not *AND*ed with another number to produce t. The function is essentially the identity function so the candidate set is $C_{t,1} = \{t\}$.

Case (2): $z(t) = 0$, $k > 1$
Since $z(t) = 0$ the target binary number is $t = 111 \cdots 1$. We require that k (at least 2) binary numbers are *AND*ed in order to produce t. Suppose these numbers exist. Also, the formulation of the problem requires that they are all distinct. Then at least one of them will have a '0' as a digit because $111 \cdots 1$ is the only number of length l with no zeros. But this implies that their image under the *AND* function will also have at least one '0' digit which contradicts the fact that the target binary number t has $z(t) = 0$. Therefore, no such k numbers can exist. Thus $C_{11\cdots 1,k} = \emptyset$ for $k > 1$.

We argue cases (3) and (4) together because they are closely related.

Case (3) $l \geq z(t) > 0 \land 2 \leq k \leq 2^{z(t)}$ and Case (4) $l \geq z(t) > 0 \land k > 2^{z(t)}$:
In both cases the target binary number has at least one '0' and we require at least 2 binary numbers to be *AND*ed in order to produce t. Only binary numbers which have at least as many '1's, and at the same positions, as the target string can achieve this. Thus the positions of the '1's are fixed and only the positions with zeros in t can have variations, i.e., 1 or 0. This explains why the cardinality of the candidate set is $2^{z(t)}$: there are $z(t)$ positions (the number of zeros) and each can independently take two values. If k exceeds the cardinality of $|C_{t,2}|$ then we are trying to find k-tuples which have a greater number of components than the total number of distinct binary numbers in $C_{t,2}$. This would force repetition in the components and this by definition is prohibited. Thus no such k-tuples can exist and $C_{t,2}$ will be empty. The only thing that remains to prove is $C_{t,k} = C_{t,2}$ if $l \geq z(t) > 0$ and $2 \leq k \leq 2^{z(t)}$. In other words, the candidate set remains invariant given that the above conditions are met.

First we show that $C_{t,k} \subseteq C_{t,2}$. Let *AND*$_k((b_1, b_2, \ldots, b_k)) = t$ for some t. Then $b_1, b_2, \ldots, b_k \in C_{t,k}$. This k-tuple though is "equivalent" to the following 2-tuples (b_1, t), (b_2, t), …, (b_k, t) since if we apply the *AND*$_2$ function to each 2-tuple the result is t, and thus all of $b_1, b_2, \ldots, b_k$ are in $C_{t,2}$.

Conversely, we show that $C_{t,k} \supseteq C_{t,2}$. Given a series of 2-tuples (b_1, b_2), (b_3, b_4), …, (b_{k-1}, b_k) which are pre-images of t under the function *AND*$_2$, and therefore $b_1, b_2, \ldots, b_k \in C_{t,2}$, we can create the following k-tuple $(b_1, b_2, \ldots, b_k)$ which is a pre-image of t under the *AND*$_k$ function. The reason for this is because bitwise *AND*ing is an associative operation. Thus $b_1, b_2, \ldots, b_k \in C_{t,k}$. Therefore we have proved that $C_{t,k} = C_{t,2}$. □

This proof reveals a very simple characterization for the candidate sets. A candidate set, in essence, comprises all the binary numbers which have '1's at the same positions as the target t and have at least as many total number '1's as t. Starting with our example target string $t = 1010$, all the elements in $C_{1010,2}$ will have the form 1␣1␣ where ␣ could be 1 or 0. More specifically $C_{1010,2} = \{1010, 1011, 1110, 1111\}$. This explains why $11 \cdots 1$ appears in all the candidate sets $C_{t,2}$ (except its own, i.e., $C_{11\cdots 1,2}$), whereas, $00 \cdots 0$ appears only in its own candidate set. And this also implies that the target binary number will always be an element of its own candidate set, and actually the smallest such element, i.e., $t = \min\{C_{t,k}\}$ (other elements will have one or more '1's in positions that have '0's in t, and thus will be larger than t). This puts a lower bound of $\Omega(2^{z(t)})$ on the creation of a specific candidate set. This is because one must spend $2^{z(t)}$ time to create all of the $2^{z(t)}$ combinations.

Corollary 1.

$$|C_{t,k}| = \begin{cases} 1 & , \quad k = 1 \\ 0 & , \quad z(t) = 0 \land k > 1 \\ 2^{z(t)} & , \quad l \geq z(t) > 0 \land 2 \leq k \leq 2^{z(t)} \\ 0 & , \quad l \geq z(t) > 0 \land k > 2^{z(t)} \end{cases}$$

Proof. This follows directly from Theorem 1. □

For example, with our target bit pattern of $t = 1010$, we have $z(t) = 2$ and the *candidate set* is $C_{1010,2} = \{10, 11, 14, 15\}$ with $|C_{1010,2}| = 2^2 = 4$.

We define the *summary set* as the set of all candidate sets of all binary numbers of length l.

$$S_{l,k} = \{C_{t,k} : \forall t \in \mathbb{B} \text{ s.t. } |t| = l\}$$

For $l = 4$ and $k = 2$, the last column in the table below provides the elements of $S_{4,2}$.

Binary Number t	$\|C_{t,2}\|$	$C_{t,2}$
0000	16	$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\}$
0001	8	$\{1, 3, 5, 7, 9, 11, 13, 15\}$
0010	8	$\{2, 3, 6, 7, 10, 11, 14, 15\}$
0011	4	$\{3, 7, 11, 15\}$
0100	8	$\{4, 5, 6, 7, 12, 13, 14, 15\}$
0101	4	$\{5, 7, 13, 15\}$
0110	4	$\{6, 7, 14, 15\}$
0111	2	$\{7, 15\}$
1000	8	$\{8, 9, 10, 11, 12, 13, 14, 15\}$
1001	4	$\{9, 11, 13, 15\}$
1010	4	$\{10, 11, 14, 15\}$
1011	2	$\{11, 15\}$
1100	4	$\{12, 13, 14, 15\}$
1101	2	$\{13, 15\}$
1110	2	$\{14, 15\}$
1111	0	$\emptyset$

4 Properties of Candidate Sets

We now show three useful properties of candidate sets. The first concerns the size of these sets.

Lemma 1. *The average cardinality of the candidate sets for $k = 2$ and for a given l is $\overline{|C|} = \frac{3^l - 1}{2^l}$.*

Proof. The average is $\overline{|C|} = \dfrac{\left(\sum_{z=0}^{l} \binom{l}{z} \cdot 2^z\right) - 1}{2^l}$.
$\sum_{z=0}^{l} \binom{l}{z} \cdot 2^z$ is the binomial expansion of $(2+1)^l = 3^l$. So $\overline{|C|} = \frac{3^l - 1}{2^l}$. □

Note that $\overline{|C|} = \frac{3^l - 1}{2^l} < 1.5^l = O(1.5^l)$. For $l = 10$ a candidate set will contain on average about 5% of the possible binary numbers of length l. For $l > 20$ a candidate set will contain on average only about 0.3% of the possible strings. This is expected since the fraction $\frac{1.5^l}{2^l}$ decreases as l increases. Note also that $\overline{|C|}$ is an upper bound for the mean of the cardinalities of all the elements in $S_{l,k}$, where $k > 2$. This is because all the elements in $S_{l,2}$ have their maximum achievable cardinality, and as k increases more and more elements in $S_{l,k}$ become empty. For example, initially $C_{1010,2} = \{10, 11, 14, 15\}$ but $C_{1010,k\geq 5} = \emptyset$.

This decrease in candidate set cardinality as l increases has implications for forensic analysis. Recall that the goal is to determine the set of possible corruption events implied by a provided target binary number. While the number of possibilities grows as l gets larger, the *percentage* of possible granules declines.

The second property concerns the elements of the summary set: for $k = 2$, there are 2^l candidate sets of the binary numbers of length l, i.e., $|S_{l,2}| = 2^l$. This is a direct result (under the assumptions on k and l) of the uniqueness of candidate sets.

Lemma 2. *For $k = 2$, the candidate sets of all the binary numbers of length l are unique.*

Proof. Case (1) We have $C_{t,2}$ and $C_{t',2}$ with $t \neq t'$ and $z(t') \neq z(t)$ where $z(t)$ and $z(t')$ are the number of zeros of targets t and t' respectively. Assume without loss of generality that $z(t) > z(t')$. Then, since both numbers have the same length there exists at least one position in t where t has a '0' and t' has a '1'. Since t' has a '1' at that position then *all* the numbers in its candidate set will have a '1' at that same position. This is not the case with the numbers in $C_{t,k}$ since they can have either a '0' or a '1' at that position. Therefore $C_{t,2} \neq C_{t',2}$.

Case (2) We have $C_{t,2}$ and $C_{t',2}$ with $t \neq t'$ and both t' and t have the same number of zeros $z(t)$. This implies they also have the same number of '1's since they both have the same length. However, for the two numbers to be different, there must exist at least one position in t where t has a '0' and t' has a '1'. Using the same argument as before this implies that $C_{t',2} \neq C_{t,2}$. □

Candidate sets also exhibit the following fundamental property: they are related (specifically, through set intersection) to the candidate sets of the constituent binary numbers that combine (through logical *OR*) to form the target.

Lemma 3. *Let $C_{t,k}$, $t \in \mathbb{B}$, and $a_1, a_2, \ldots, a_m \in \mathbb{B}$ s.t. $\bigvee_{j=1}^{m} a_j = t$ for some $m \leq 2^{|t|}$ and let also $2 \leq k \leq 2^{z(t)}$. Then:*

$$C_{t,k} = C_{\bigvee_{j=1}^{m} a_j,k} = \bigcap_{j=1}^{m} C_{a_j,k}$$

Proof. Forward direction $\Longrightarrow$: Let $b_1, b_2, \ldots, b_k \in C_{t,k}$. We need to show that $b_1, b_2, \ldots, b_k \in \bigcap_{j=1}^{m} C_{a_j,k}$. By definition we know that $b_1 \wedge b_2 \wedge \ldots \wedge b_k = t$. However, we are also given that $\bigvee_{j=1}^{m} a_j = t$. Thus, $\bigvee_{j=1}^{m} a_j = t = \bigwedge_{i=1}^{k} b_i$. Therefore, we must prove that for every b_i ($1 \leq i \leq k$) there exists a series of $k-1$ distinct binary numbers (and different from b_i), $d_1, d_2, \ldots, d_{k-1}$ such that $b_i \wedge d_1 \wedge d_2 \wedge \ldots \wedge d_{k-1} = a_j \Rightarrow b_i, d_1, d_2, \ldots, d_{k-1} \in C_{a_j,k}$ for each $a_j, 1 \leq j \leq m$. In other words, each one of the b_is must appear in the pre-image of each one of the a_js.

We proceed to show how to produce all the requisite $b_i, d_1, \ldots, d_{k-1}$ given a specific b_i and a_j pair. Let x be the number of '1's in the binary number t, y be the number of '1's in a specific b_i, and w the number of '1's in a specific a_j. Then $y \geq x$ since b_i must have at least the same number of '1's, and at the same positions, as the target number t. This is true for all b_i since for a '1' to appear at a specific position in t then *all* the binary numbers b_i, which when *AND*ed produce t, must have a '1' at the same position. Likewise, $x \geq w$ since a_j must have at most the same number of '1's as the target number t. Again, this is true for all a_j since for a '1' to be preserved at a specific position in t at least one of the a_j must have a '1' at that same position. Using the observation above we begin with some b_i and pick d_1 to be a_j. This works because we want a number d_1 which has a zero at the same positions as a_j does, in order to mask any '1's b_i has at those positions. d_1 should also have a '1' wherever a_j does, so that the '1'is preserved after the *AND* operation. Note that if a_j has a '1' at a certain position we are guaranteed to have a '1' at the same position in b_i because t will have a '1' at that position (as discussed previously). All the rest of the $k-2$ binary numbers can be created from a_j and there are enough of them: $2^{z(a_j)} - 1$ (the '-1' is there because we are excluding a_j itself) where $z(a_j)$ is the number of zeros in a_j. We are given that $k \leq 2^{z(t)}$ and since $w + z(a_j) = x + z(t) = l$ and $x \geq w$ then $z(t) \leq z(a_j)$. Thus, $k-2 < k \leq 2^{z(t)} \leq 2^{z(a_j)} \Rightarrow k-2 \leq 2^{z(a_j)} - 1$. This implies that each of the b_i is an element of each of the $C_{a_j,k}$ and therefore an element of their intersection. Thus, $C_{\bigvee_{j=1}^{m} a_j,k} \subseteq \bigcap_{j=1}^{m} C_{a_j,k}$.

Backward direction $\Longleftarrow$: Conversely, let $b \in \bigcap_{j=1}^{m} C_{a_j,k}$. Then $(b \in C_{a_1,k}) \wedge (b \in C_{a_2,k}) \wedge \ldots (b \in C_{a_m,k})$. This implies that b has a '1' at the same positions as a_1, b has a '1' at the same positions as a_2 and so on until a_m. Thus the fact that b belongs to all the candidate sets of the a_is, fixes the positions of the '1's while the remaining positions could be '0' or '1'. Thus b captures a certain set of numbers. Now, consider $\bigvee_{j=1}^{m} a_j = t$. We know that t, as a result of an *OR* operation, will have a '1' wherever at least one a_i has a '1' at that position, and a '0' wherever all a_is have a '0' at that position. The candidate set of target t comprises all the numbers which have a '1' at the same position as t and at least as many '1's as t, i.e., wherever t has a '0' the pre-images can have a '0' or a '1'. But this is exactly the same set of numbers captured by b so $b \in C_{t,k}$. Therefore, $C_{\bigvee_{j=1}^{m} a_j,k} \supseteq \bigcap_{j=1}^{m} C_{a_j,k}$. □

This provides a pleasing symmetry between the logical *AND* in the definition of the candidate set and the logical *OR* used above to form the target.

We now turn to ways in which the candidate set may be computed. We first give an algorithm that is optimal in time, except for a very few cases. Following some further observations on the summary set, we show how, given a candidate set, one can calculate all summary sets with a smaller l in constant time.

5 Computing the Candidate Set

We now give an optimal algorithm for computing the candidate set given the target string t and k, and again assuming a fixed l. This algorithm generates the elements in the candidate set in numeric order, interestingly, using a linear-time sort. All arrays and strings use zero-based indexing. All parameters are passed by value (the C code uses the more efficient pass-by-reference for arrays and strings).

```
// input:   a binary target number $t$, its length $l$, and a function index $k$ for *AND*$_k$
// output:  $C_{t,k}$, an array of binary numbers (also created is an array of zeros, *Z*)
1:    candidateSet(unsigned int $t$, int $l$, int $k$)
2:          $C_{t,k} \leftarrow$ new array()
3:          $z \leftarrow 0$
4:          *Z* $\leftarrow$ new array()
5:          **for** $i \leftarrow l - 1$ **to** $0$
6:                **if** $t \,\&\, (1 << i) = 0$ **then** $z \leftarrow z + 1$
7:                $Z[l - i - 1] \leftarrow z$
8:          **if** $k < 1 \vee k > 2^l$ **then report** NOT_DEFINED
9:          **else if** $k = 1$ **then** $C_{t,k} \leftarrow \{t\}$
10:         **else if** $(z = 0 \wedge k > 1) \vee (l \geq z > 0 \wedge k > 2^z)$ **then** $C_{t,k} \leftarrow \emptyset$
11:         **else if** $(l \geq z > 0) \wedge (2 \leq k \leq 2^z)$ **then**
12:               *rightmost* $\leftarrow$ createRightmost($t$, $l$)
13:               $C_{t,k} \leftarrow$ generate($t$, *rightmost*$[l]$, $l$, $C_{t,k}$)
14:               $C_{t,k} \leftarrow$ funkySort($z$, $C_{t,k}$)
15:         **return** $C_{t,k}$

// fill in array $C_{t,k}$
1:    generate(unsigned int $t$, int $p$, int $l$, array $C_{t,k}$)
2:          **if** $p = -1$ **then** $C_{t,k}$.append($t$)
3:          **else**
4:                $C_{t,k} \leftarrow$ generate($t$, *rightmost*$[p]$, $l$, $C_{t,k}$)
5:                $C_{t,k} \leftarrow$ generate($t + (1 << (l - p - 1))$, *rightmost*$[p]$, $l$, $C_{t,k}$)
6:          **return** $C_{t,k}$

// fill in array *rightmost*
1:    createRightmost(unsigned int $t$, int $l$)
2:          int $i, j,$ *flag*
3:          $j \leftarrow -1$
4:          *flag* $\leftarrow$ FALSE
5:          *rightmost* $\leftarrow$ new array()
6:          **for** $i \leftarrow l - 1$ **to** $-1$
7:                **if** *flag* $=$ *TRUE* **then** $j \leftarrow l - i - 2$
8:                **if** $t \,\&\, (1 << i) = 0$ **then** *flag* $\leftarrow$ TRUE
9:                **else** *flag* $\leftarrow$ FALSE
10:               *rightmost*$[l - i - 1] \leftarrow j$
11:         **return** *rightmost*

// input:   $z$, the number of zeros
//          $C_{t,k}$ an unsorted array
// output:  $C_{t,k}$ in ascending order
1:    funkySort(int $z$, array $C_{t,k}$)
2:          *sorted* $\leftarrow$ new array()
3:          *indices* $\leftarrow$ new array()
4:          *indices*$[0] \leftarrow 0$
5:          int $i,$ *offset*, *power*
6:          *offset* $\leftarrow 0$
```

```
7:      power ← $1 << z$
8:      for $i \leftarrow 1$ to $(1 << z) - 1$
9:          if $(i \,\&\, (i-1)) = 0$ then
10:             power ← power $>> 1$
11:             offset ← $0$
12:         indices[$i$] ← indices[offset] + power
13:         offset ← offset + $1$
14:     for $i \leftarrow 0$ to $(1 << z) - 1$
15:         sorted[$i$] ← $C_{t,k}$.get(indices[$i$])
16:     return sorted
```

Let us now briefly examine this algorithm. We first start by looking at the candidateSet function and discuss each different function as we encounter it.

- The use of the Z array on lines 4 and 7 will be explained later in the discussion following the proof of Theorem 2.
- Lines 8–11 follow the result of Theorem 1.
- Then on line 12 the createRightmost helper function is called to preprocess the target binary number t and to fill the *rightmost* array in order to answer the "rightmost zero" query in constant time. More specifically, *rightmost*$[p]$ is the index (bit position) of the rightmost zero to the left of index p non-inclusive. Within this function i iterates over t from left to right (high-order to low-order bits). The flag is required because we must remember what we saw in the previous iteration: if *flag* $=$ TRUE we saw a 0, otherwise we saw a 1. This runs in $\Theta(l)$.
- On line 13 the generate function is called. This is a recursive function which creates the candidate set elements. Given a position p, which is a specific index in the zero-based enumeration (left to right) of the binary number t, it finds the index of the rightmost zero to the left of p using the *rightmost* array. It first recurses on that index maintaining the same binary number (line 4) and then sets the digit at position p to 1 and recurses on the same index *rightmost*$[p]$ but with this new number (line 5). We can consider the input target string t as capturing all the $2^{z(t)}$ number that must be generated during the recursion, so we can consider the input size to be $n = 2^{z(t)}$. Also, at each recursive call the position of the zero processed is never revisited so the input size at each call is essentially halved. Moreover, the amount of work done at each stage of the recursion is constant hence the formula that captures this recursion is $T(n) = 2T(\frac{n}{2}) + \Theta(1)$. The solution of this formula is $\Theta(n)$ so the running time of the generate function is $\Theta(2^{z(t)})$. However, a side-effect of this recursive creation of the candidate set elements is that the elements are not in numeric order.
- On line 14 we call the sorting function. Even though the elements are not sorted there does exist a pattern in the order in which they are created. This funkySort function creates the sequence of indices which when used to index into the $C_{t,k}$ array will result in the ordering of the candidate set elements. This is achieved by performing a single pass over the *indices* array and creating each new index by manipulating appropriately previous ones (lines 8–13) within the funkySort function. For example, with a target string of $t = 10000$, i.e., 16 in decimal, after the generate function finishes the candidate set will be $C_{t,k} = \{16, 24, 20, 28, 30, 17, 25, 21, 29, 19, 27, 23, 31\}$ in this order. Examining closely the set we see that in order to create the *sorted* array we must recursively visit the first element of each subsequent half of $C_{t,k}$. Line 12 creates this sequence of indices: 0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15. More specifically, by starting from 0 the 8 can be created by $0 + 2^{z(t)}$ where $z(t) = 3$. Then, 4 and 12 can be obtained by adding 2^2 to each of 0 and 8. Then 2, 10, 6 and 14 are obtained by adding 2^1 to 0, 8, 4, 12 respectively. Finally, the last 8 numbers are obtained by adding 2^0 to the first 8 numbers. This explains why at elements appearing at indices which are powers of 2 in the *indices* array, the *offset* is reset to zero and the *power* is halved. On lines 14–15 we use the sequence of indices we created and the actual sorting happens. Note that on line 9 even though the test $(i \,\&\, (i-1)) = 0$ accepts 0 as a power of 2 in this case this is exactly what we want. This pass over the *indices* array runs in $\Theta(2^{z(t)})$.

The running time of candidateSet is $\Theta(l + 2^{z(t)})$. Thus the algorithm is optimal most of the time, using the lower bound given earlier, except for the very few cases when $l > 2^{z(t)}$. In terms of space complexity the algorithm given here requires $O(l + 2^{z(t)})$ space, as does the C code implementation given in the appendix, which requires $l + \max\{l, 2^{z(t)}\} + 2^{z(t)}$ space.

6 A Second Algorithm, Given a Summary Set

We now show that for fixed k and given $S_{l,k}$ one can calculate all $S_{l',k}$ s.t. $l' < l$ without resorting to the algorithm given previously. The technique shown below can be potentially faster. We define the *candidate array*, denoted $A_{t,k}$, to be an array which contains the elements of $C_{t,k}$ sorted in ascending numerical value. Then, $A_{t,k}[x : y]$ selects all elements in the candidate array from index x to y. (NB: $A_{t,k}[i] = A_{t,k}[i : i]$). Also, for reasons of ease and precision we annotate A with the length of the binary number whose value was previously implicit, as a leading subscript.

Given a candidate array ${}_lA_{y,k}$ for a specific target string y, we wish to compute the candidate array ${}_{l-x}A_{t,k}$ where t is a suffix ($l = |y| > l - x = |t| \geq 1$) of y. Each $S_{l,k}$ captures all the candidate sets for all $l' < l$. This method creates each element of $S_{l',k}$ by exploiting the fact that each of the binary numbers of length l' is a suffix of more than one corresponding binary number of length l . For example, the candidate set $C_{1010,2}$ can be computed from the candidate sets of 01010, 001010, 101010 and so on. Let $y = p \bullet t = \{0,1\}^x t$ for some prefix p of length x. Let *Suffix*$_i(s)$ denote the suffix of string s starting at position i.

Let us look at some examples to develop some intuition. Given ${}_4A_{0010,2} = [0010, 0011, 0110, 0111, 1010,$ $1011, 1110, 1111]$ we wish to compute ${}_3A_{010,2}$. Observe that $t = 010$ is $y = 0010$ with the leftmost '0' removed. Removing the leading '0' from y results in a string t which cannot encode any numbers in the range 2^3 to $2^4 - 1$. Thus the candidate array of ${}_3A_{010,2}$ will have the same elements as the candidate array of ${}_4A_{0110,2}$ except for the numbers encoded by the extra leading digit. We know that each additional '0' present in the target string doubles the cardinality of the candidate set, thus a removal of the zero will halve the number of candidate set elements. Observe also that the elements in the second half of ${}_4A_{0010,2}$ have essentially the same bit pattern as the elements in the first half but with a '1' at the leftmost position instead of a '0', e.g., 1110 has the same bit pattern as 0110 apart from the bit in the leftmost position. Thus in order to compute ${}_3A_{010,2}$ we can truncate the leftmost digit from all the elements in the original candidate set. By removing the leftmost digit from each of the elements in ${}_4A_{0010,2}$, we get 010, 011, 110, 111, 010, 011, 110, 111. The first half of the elements will have a leading '0' removed, something which will not change their numerical value, while the second half which will have a leading '1' removed will produce identical numbers of length 3 to the truncated numbers in the first half. Since the cardinality of ${}_3A_{010,2}$ is half that of ${}_4A_{0010,2}$, and since the two halves of ${}_4A_{0010,2}$ have the same elements after the truncation and by knowing that ${}_3A_{010,2} = [010, 011, 110, 111]$ we can verify that:

$$\begin{aligned} {}_3A_{010,2} &= [\mathit{Suffix}_1({}_4A_{0010,2}[0]), \mathit{Suffix}_1({}_4A_{0010,2}[1]), \mathit{Suffix}_1({}_4A_{0010,2}[2]), \mathit{Suffix}_1({}_4A_{0010,2}[3])] \\ &= [010, 011, 110, 111] = [2, 3, 6, 7] \end{aligned}$$

Let us consider a different example in which the original target string is $y = 1010$ and the same suffix $t = 010$. In this case ${}_4A_{1010,2} = [1010, 1011, 1110, 1111]$ all elements necessarily start with a '1'. Since removing the leading '1' from y to get t does not affect the number of zeros in the strings the cardinalities of the two candidate set is the same. Removing the leftmost '1' from all the elements of ${}_4A_{1010,2}$ will yield directly the desired elements of the new candidate set:

$$\begin{aligned} {}_3A_{010,2} &= [\mathit{Suffix}_1({}_4A_{1010,2}[0]), \mathit{Suffix}_1({}_4A_{1010,2}[1]), \mathit{Suffix}_1({}_4A_{1010,2}[2]), \mathit{Suffix}_1({}_4A_{1010,2}[3])] \\ &= [010, 011, 110, 111] = [2, 3, 6, 7] \end{aligned}$$

With these valuable observations we can now state the theorem and its proof.

Theorem 2. *Assume $y = p \bullet t = \{0,1\}^x t$, $0 < x < l$, $0 \leq z(t) \leq l - x$ and $q = 2^{z(t)}$. Then:*

$$ {}_{l-x}A_{t,k} = \begin{cases} \text{N/A} & , \quad k > 2^{l-x} \qquad (1) \\ [t] & , \quad k = 1 \qquad (2) \\ \emptyset & , \quad z(t) = 0 \wedge 1 < k \leq 2^{l-x} \qquad (3) \\ \bigcup_{0 \leq i < q} [\mathit{Suffix}_x({}_lA_{y,2}[i])] & , \quad l - x \geq z(t) > 0 \wedge 2 \leq k \leq q \qquad (4) \\ \emptyset & , \quad l - x \geq z(t) > 0 \wedge k > q \qquad (5) \end{cases}$$

Proof. Case (1):
The candidate set is not defined when we try to deduce a candidate set for a binary number of length $l-x$ given that the (original) k is greater than the total number of possible numbers that can be created using $l-x$ digits, i.e., 2^{l-x}. This is true since as discussed at the beginning of the paper this would force repetition of a binary number.

Case (2), Case (3) and Case (5):
These follow directly from the proof of Theorem 1.

Case (4):
It is worth elucidating here the nature of the number q. This number can be thought of as the cardinality of the candidate set of the suffix t: $q = 2^{z(t)}$ according to Corollary 1. It can alternatively be defined as $q = \frac{1}{2^{z(p)}}|{}_lC_{y,2}|$, that is, it is the cardinality of the candidate set of the original target y scaled down by a power of 2. This power of 2 is given by the number of zeros present in the truncated prefix p. Regarding q in this respect is consistent with the its initial assumption as $q = 2^{z(t)}$. This is true since $y = \{0,1\}^x t \Rightarrow z(y) = z(p) + z(t)$, which in turn implies that $q = \frac{1}{2^{z(p)}}|{}_lC_{y,2}| = \frac{2^{z(y)}}{2^{z(p)}} = 2^{z(t)}$.

We prove case (4) by induction on x. Define proposition:
$P(x) : {}_{l-x}A_{t,k} = \bigcup_{0 \le i < q}[\textit{Suffix}_x({}_lA_{y,2}[i])]$ for $(l-x \ge z(t) > 0) \wedge (2 \le k \le 2^{z(t)})$ and $q = 2^{z(t)}$.

Basis of induction: Prove $P(1)$ is true.
Let $x = 1$. Here the prefix p is a single bit. We have that $q = \frac{1}{2^{z(\{0,1\})}}|{}_lC_{y,2}| = 2^{z(t)}$, $y = \{0,1\} \bullet t$ and we want to prove that ${}_{l-1}A_{t,k} = \bigcup_{0 \le i < q}[\textit{Suffix}_1({}_lA_{y,2}[i])]$.
Thus, ${}_{l-1}A_{t,k} = [\textit{Suffix}_1({}_lA_{y,2}[1]), \textit{Suffix}_1({}_lA_{y,2}[2]), \ldots, \textit{Suffix}_1({}_lA_{y,2}[q])]$. What $P(1)$ essentially claims is that the new candidate array ${}_{l-x}A_{t,k}$ can be computed by simply selecting the first q elements of the candidate array ${}_lA_{y,k}$ and removing the leftmost digit from each such element selected.

Case (i) Assume that $p = 0$ (this corresponds to the first example, above). With respect to this first digit of the target binary string y we can divide the elements of its candidate array into two groups: those which have a '1', and those which have a '0' at that leftmost position. Due to the way these elements are created, resulting in the elements of the candidate array being sorted in increasing order, the elements with a '1' for a leftmost digit must all appear after those with a '0' at the same position. Depending upon its position, each digit encodes the numbers in the range 2^{i-1} to $2^i - 1$ where i ($1 \le i \le l$) is the position of the digit numbering the string from right to left. So by removing the leading '0' from y results in a string t which cannot encode any numbers in the range 2^{l-1} to $2^l - 1$. Thus the candidate array of ${}_{l-1}A_{t,k}$ will have the same elements as the candidate array of ${}_lA_{y,k} = {}_lA_{y,2}$ except for the numbers encoded by the extra leading digit. But we know that each additional '0' introduced doubles (the position can be filled by a '0' or a '1') the count of numbers that can be encoded which implies removing a '0' will halve the count of numbers encoded: $z(p) = 1 \Rightarrow z(t) = z(y) - 1 \Rightarrow |{}_{l-1}C_{t,k}| = 2^{z(t)} = 2^{z(y)-1} = \frac{1}{2}|{}_lC_{y,k}|$. Thus the two groups of elements mentioned in the beginning will be equinumerous: the elements in the second half have essentially the same bit pattern as the elements in the first half but with a '1' at the leftmost position instead of a '0'. By removing the leftmost digit from each of the elements in ${}_lA_{y,k}$, the first half will have a leading '0' removed, something which will not change their numerical value, while the second half which will have a leading '1' removed will produce identical numbers of length $l-1$ to the truncated numbers in the first half. This is the reason why ${}_{l-1}A_{t,k}$ will comprise the suffixes starting at position 1, of the elements in the first half (i.e., $\frac{1}{2^1}|{}_lC_{y,2}| = q$) of the numbers in the array ${}_lA_{y,2}$.

Case (ii) Assume that $p = 1$ (this corresponds to the second example, above). In this case the situation is simpler since all the elements in ${}_lA_{y,k}$ can only start with a '1'. Since the number of zeros in t remains unaltered ($z(p) = 0 \Rightarrow z(y) = z(t)$), this implies that $|{}_{l-1}C_{t,k}| = |{}_lC_{y,k}|$. Thus removing the leftmost digit from all the elements of ${}_lA_{y,k}$ will yield directly the desired elements of the new candidate set since each of the truncated elements will have the same numerical value as their binary number counterparts of length l with a '0' at the leftmost position. Again the new candidate array ${}_{l-1}A_{t,k}$ will comprise the suffixes starting at position 1, of the q ($= \frac{1}{2^0}|{}_{l-1}C_{t,k}|$) first elements (in this case all of them) of ${}_lA_{y,k}$.

Inductive step: Prove that $P(x) \longrightarrow P(x+1)$
We assume that ${}_{l-x}A_{t,k} = \bigcup_{0 \le i < q}[\textit{Suffix}_x({}_lA_{y,2}[i])]$, where $q = \frac{1}{2^{z(p)}}|{}_lC_{y,2}|$, and $y = p \bullet t = \{0,1\}^x t$ is true and seek to use this inductive hypothesis to prove ${}_{l-(x+1)}A_{t',k} = \bigcup_{0 \le i < q'}[\textit{Suffix}_{x+1}({}_lA_{y,2}[i])]$ where $\{0,1\}t' =$

$t \Rightarrow y = \{0,1\}^x t = \{0,1\}^x \{0,1\} t' = \{0,1\}^{x+1} t'$, and $q' = \frac{1}{2^{z(p)+z(\{0,1\})}} |{}_l C_{y,2}|$. Thus:

$$\begin{aligned}
{}_{l-(x+1)}A_{t',k} &= {}_{(l-x)-1}A_{t',k} && \text{apply basis of induction} \\
&= \bigcup_{0 \le i < \hat{q}} [\mathit{Suffix}_1({}_{l-x}A_{\{0,1\}t',2}[i])] && \text{where } \hat{q} = \frac{1}{2^{z(\{0,1\})}} |{}_{l-x}C_{\{0,1\}t',2}| \\
&= \bigcup_{0 \le i < \hat{q}} [\mathit{Suffix}_1({}_{l-x}A_{\{0,1\}t',k}[i])] && \text{candidate set is invariant when } k \le 2^{z(t')} \\
&= \bigcup_{0 \le i < \hat{q}} [\mathit{Suffix}_1({}_{l-x}A_{t,k}[i])] && \text{since } \{0,1\} \bullet t' = t \\
&= \bigcup_{0 \le i < \hat{q}} [\mathit{Suffix}_1\big(\big(\bigcup_{0 \le j < q} [\mathit{Suffix}_x({}_lA_{y,2}[j])]\big)[i]\big)] && \text{by inductive hypothesis} \\
&= \bigcup_{0 \le i < \hat{q}} [\big(\bigcup_{0 \le j < q} [\mathit{Suffix}_1(\mathit{Suffix}_x({}_lA_{y,2}[j])]\big)[i])] && \text{the suffix and union operations commute} \\
&= \bigcup_{0 \le i < q'} [\mathit{Suffix}_1(\mathit{Suffix}_x({}_lA_{y,2}[i]))] \\
&= \bigcup_{0 \le i < q'} [\mathit{Suffix}_{x+1}({}_lA_{y,2}[i])]
\end{aligned}$$

By the first principle of mathematical induction the initial proposition is true. □

The algorithm for computing the candidate sets using this new method is given below. Note that t_{start} is the index in the original string y where the suffix t starts.

```
// input:  initial candidate set $C_{y,k}$
//         bit position $t_{start}$ at which the suffix $t$ starts in $y$
//         length of original target string $l_y$
// output: the candidate set $C_{t,k}$
1:  candidateSetSuffix ( array $C_{y,k}$, int $t_{start}$, int $k$, int $l_y$, array Z )
2:      $C_{t,k} \leftarrow$ new array()
3:      $l_t \leftarrow l_y - t_{start}$
4:      $z_t \leftarrow z_y - Z[t_{start} - 1]$
5:      $mask \leftarrow (1 << l_t) - 1$
6:      $y \leftarrow C_{y,k}[0]$
7:      $t \leftarrow y \,\&\, mask$
8:      **if** $k < 1 \vee k > 2^{l_t}$ **then report** NOT_DEFINED
9:      **else if** $k = 1$ **then** $C_{t,k} \leftarrow \{t\}$
10:     **else if** $(z_t = 0 \wedge 1 < k \le 2^{l_t}) \vee (l_t \ge z_t > 0 \wedge k > 2^{z_t})$ **then** $C_{t,k} \leftarrow \emptyset$
11:     **else if** $(l_t \ge z_t > 0) \wedge (2 \le k \le 2^{z_t})$ **then**
12:         **for** $i \leftarrow 0$ **to** $2^{z(t)} - 1$
13:             $C_{t,k}$.append($C_{y,k}[i] \,\&\, mask$)
14:     **return** $C_{t,k}$
```

Since creating the candidate set for y involves scanning all of y to find the zeros we can at the same time maintain an array which accumulates the number of zeros encountered so far during the scan. This array is the Z array which was created in the function candidateSet (lines 4, 7). We can index into this array using the position which suffix t starts in y and thus get the number of zeros in constant time. For example, for $y = 01101010$ and $t = 1010$ given in terms of t_{start} which is the start position of t in y, we can scan y from left to right and create the array $Z = [1,1,1,2,2,3,3,4]$. This arrays gives the number zeros in every suffix of y. Thus, $z(t) = z(y) - Z[t_{start} - 1]$. In this case $t_{start} = 4$, and so $z(1010) = z(01101010) - Z[4-1] = 4 - 2 = 2$.

In addition, the *mask* is used as a means of setting the first x bits of each original candidate set element to zero, which is the equivalent in a sense of taking the suffix of the corresponding binary string. For example, if the candidate set element is 18, with binary representation 10010, and we want to take the suffix starting at index

2, then the *mask* $= 7$ (00111 in binary). Thus, by bitwise *AND*ing the *mask* and the element, we get $010 = 2$. Note that the masking does not simply set the higher order bits to zero but it truncates the number, i.e. the length actually decreases. This is important because we seek to derive from the candidate set of 10010 the candidate set of 010 and not the set for 00010. The latter is impossible to derive in the way described in this section since $C_{00010,2}$ is a *superset* of $C_{10010,2}$.

The "for" loop on line 12 dominates the running time of the above algorithm. Hence, the algorithm, in the worst case, runs in $\Theta(2^{z(t)})$, which is optimal.

However, we can do better by using a different representation for the candidate set of the suffix t. Since the elements of $C_{t,k}$ are contiguous elements of $C_{y,k}$ starting at position 0 then the candidate set of t can be given as a range of values. This is achieved just by maintaining a pointer to the position $q - 1$ in the candidate array of y marking the last element of $C_{t,k}$. Thus, only two numbers *mask*, and $q = 2^{z(t)}$, both of which can be computed in constant time, are needed to capture the candidate set of any suffix of target y. To create the *mask* we use l_t (as seen on line 5) which was computed from the input integer t_{start} on line 3. Obtaining q is easy since we have already computed $z(t)$ on line 4. Thus, the first and last elements of the candidate set for the suffix can be given as $C_{y,k}[0]\ \&$ *mask* and $C_{y,k}[q-1]\ \&$ *mask* respectively. This approach avoids the expensive "for" loop on line 12 and makes the algorithm run in $\Theta(1)$.

For this reason it is preferable to use the candidateSetSuffix algorithm in one particular situation: to find the candidate set for the suffix of y whenever we already have the candidate set for y. Consider the following examples.
For $l = 4$ we want to calculate $C_{1010,7}$ and $C_{010,3}$. $C_{1010,7} = \emptyset$ since $|C_{1010,7}| = 2^2 = 4 < k = 7$. In the case of $C_{010,3}$ we have $3 \geq z(t) = 2 > 0$ and $2^{z(t)} = 4 > k = 3$ so

$_3A_{010,3} = \bigcup_{1 \leq i \leq 4}[\mathit{Suffix}_1(_4A_{1010,2}[i])] = \bigcup_{1 \leq i \leq 4}[\mathit{Suffix}_1[1010, 1011, 1110, 1111]] = [010, 011, 110, 111]$
and thus $C_{010,3} = \{2, 3, 6, 7\}$. If we decide to use the faster constant running time approach the result will be given as *mask* $= 0111$ and $q = 2^2 = 4$ and hence the first element in $C_{010,3}$ is $_4A_{1010,2}[0]\ \&\ 0111 = 1010\ \&\ 0111 = 010 = 2$ while the last element is $_4A_{1010,2}[4-1]\ \&\ 0111 = 1111\ \&\ 0111 = 111 = 7$.

Assume that we are auditing a variety of databases, each with a particular l value (for the example in this paper, $l = 4$). Within the forensic analyzer, we could precompute a summary set for l_{max}, which is the maximum of the l values that were specified for the databases that were being audited. During forensic analysis of a specific database corruption that was detected using the polychromatic algorithm discussed in Section 1, given the resulting target string and the l value for this particular database (with $l \leq l_{max}$), this algorithm could calculate in constant time the candidate set, which consists of all the possible corrupted granules that could have yielded that target number for that value of l.

7 Previous Work

Elsewhere we have introduced the approach of using cryptographic hash functions to detect database tampering [5] and of introducing additional hash chains to improve forensic analysis [2].

Strachey has considered table lookup to increase the efficiency of bitwise operations [6]. He provides a logarithmic time/logarithmic space algorithm for reversing the bits in a word. Our second algorithm requires only constant time, but the table must be of exponential space.

Enumerating all solutions (pre-images) is a key step in formal verification. Sheng and others have developed efficient pre-image computation algorithms [4, 1]. These algorithms are similar to the ones introduced in this paper in that they all enumerate all possible solutions. The formal verification algorithms differ in that they are computing pre-images of a state transition network, rather than of bitwise *AND* functions, as in our paper.

8 Summary

We have developed a method that allows us to find all the database partition granules that might have been corrupted when that database has been tampered. To this end we have used the notion of a candidate set which is associated with a target binary number obtained from the polychromatic forensic analysis algorithm; this candidate set captures all the potentially corrupted granules. We observed that the candidate set comprises all the pre-images (granules) of the target number, under an appropriate bitwise *AND* function specified by an index k. We have analyzed completely the behavior of the candidate sets, depending on k and the target binary number, and then developed an optimal algorithm to produce these candidate sets. We then introduced a constant-time algorithm which is preferable in the case when the target binary number is a suffix of another binary number for which a candidate set already exists. We provided proofs of correctness for both algorithms and a thorough space and time complexity analysis. These algorithms compute the possible database corruptions given information on which hash chains matched those computed before the tampering, thereby providing important information on what data was tampered with, helping to identify who did the tampering and why.

References

[1] B. Li, M. S. Hsiao, and S. Sheng, "A Novel SAT All-Solutions Solver for Efficient Preimage Computation," in *Proceedings of the IEEE International Conference on Design, Automation and Test in Europe*, Volume 1, February 2004.

[2] K. E. Pavlou and R. T. Snodgrass, "Forensic Analysis of Database Tampering," in *Proceedings of the ACM SIGMOD International Conference on Management of Data* (SIGMOD), pp. 109–120, Chicago, June, 2006.

[3] K. E. Pavlou, R. T. Snodgrass, and S. S. Yao, "Detection and Forensic Analysis of Database Tampering," TIMECENTER Technical Report, forthcoming.

[4] S. Sheng and M. S. Hsiao, "Efficient Preimage Computation Using A Novel Success-Driven ATPG," in *Proceedings of the IEEE International Conference on Design, Automation and Test in Europe*, Volume 1, March 2003.

[5] R. T. Snodgrass, S. S. Yao, and C. Collberg, "Tamper Detection in Audit Logs," in *Proceedings of the International Conference on Very Large Databases*, pp. 504–515, Toronto, Canada, September 2004.

[6] C. Strachey, "Bitwise operations," *Communications of the ACM* 4(3):146, March 1961.

Appendix: C Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>      // compile using -lm
#define FALSE 0
#define TRUE 1
#define NOT_DEFINED -999
#define GENERATE -1

// Function Prototypes
void  generate(unsigned int i, int pos);
int find_rightmost(unsigned int i, int p);
void *mymalloc(int size);
void funky_sort(void);
void create_rightmost(unsigned int target_number);
void candidateSet(int k, unsigned int target_number, int length);
void candidateSetSuffix(int suffix_start, int length, int k, int z, int *Zarray, int *candidates);

// Globals
int length;            // the length of the target binary number
int *candidates;       // this array plays a triple role: first it is used in the
                       // create_rightmost function; then it holds the sequence of
                       // indices of the aux array that when traversed would yield
                       // the sorted candidate set elements. In the end it holds the
                       // sorted candidate set elements
int *aux;              // this array holds temporarily the unsorted candidate set
                       // elements which are created by the generate function
int tail_index = 0;    // this maintains the logical size of the aux array
int logical_size = 0;  // logical size of candidates array
int z=0;               // this is the number of zeros in the target binary number
int *Zarray;           // this Zarray[i] gives you the the number of zeros seen to
                       //  the left of target_number[i] inclusive
// Total RT: O(length + 2ˆz) -- this is optimal most of the time except in the
// cases when l > 2ˆz
int main(int argc, char *argv[])
{
    if(argc != 5)
    {
        fprintf(stderr, "Usage: targetbinarynumber length numcomponents suffixstart\n");
        fprintf(stderr, "Example:\n \t 10 4 5 3\nPlease retry!\n");
        exit(1);
    }
    unsigned int target_number = atoi(argv[1]);    // the numerical representation
                                                   // of the binary target number in decimal
    length = atoi(argv[2]);                        // the length of the target number
    int k = atoi(argv[3]);                         // k is the number of components
                                                   // in the tuple; k takes values
                                                   // between 1 and 2ˆ(l)
    int suffix_start = atoi(argv[4]);              // the bit position /index at which
                                                   // the suffix of the target number begins

    if(suffix_start < 0 || length < 1 || k < 0 || target_number < 0)
    {
        fprintf(stderr, "Negative input is not allowed, and length must be at least 1\n");
        exit(1);
    }

    if(suffix_start < 0 || suffix_start >= length) // check if the starting index for
```

```
                                                // the suffix is valid
    {
        fprintf(stderr, "suffixstart must be between 0 and %d\n", length-1);
        exit(1);
    }

    int min_length = (target_number & (target_number - 1)) ? (int)ceil(log(target_number)/log(2))
                                                  : (int)(1 + ceil(log(target_number)/log(2)));
    if(min_length > length)
    {
        fprintf(stderr, "The length of the targetbinarynumber must be at least %d\n", min_length);
        exit(1);
    }

    candidateSet(k, target_number, length);        // create and output the candidate set
                                                // of target_number
    candidateSetSuffix(suffix_start, length, k, z, Zarray, candidates); // create and
                                                // output the andidate set of the suffix
    free(aux);
    free(candidates);
    free(Zarray);
    return 0;
}

// This function creates and outputs the candidate set of the original target_number
void candidateSet(int k, unsigned int target_number, int length)
{
    int i;  // iteration variable
    // find z the number of zeros in the input target_number.
    // at the same time create the Zarray: Zarray[i] gives you the number of zeros
    // seen to the left of target_number[i] inclusive
    Zarray = (int *)mymalloc(length * sizeof(int));
    for(i = length-1; i >= 0; i--)                // O(length)
    {
        if((target_number & (1<<i)) == 0)
            z++;
        Zarray[length-i-1] = z;
    }                                           // note that Zarray[length-1] equals z

    // depending on the role it plays the candidates array needs different capacities.
    int max_cap = (1 << z) > length ? (1 << z) : length;

    candidates = (int *)mymalloc((max_cap + 1)*sizeof(int));
    aux = (int *)mymalloc((1<<z)*sizeof(int));
    if ((k < 1) || (k > (1 << length) ))
        candidates[0] = NOT_DEFINED;
    else if (k == 1)
    {
        logical_size = 1;
        candidates[0] = target_number;
    }
    else if ((z == 0 && k > 1) || (length >= z && z > 0 && k > (1 << z)))
        logical_size = 0;
    else if (length >= z && z >0 && 2 <= k && k <= (1 << z))
    {
        logical_size = 1 << z;
        // preprocessing required to answer "rightmost zero" query in constant time
        create_rightmost(target_number);            // O(length)
        generate(target_number, candidates[length]); // generate the candidate set
```

```
                                                      // elements recursively in
                                                      // O(2ˆz) time
        funky_sort();                                 // sort the generated candidate
                                                      // set elements O(2ˆz)
    }
    if(candidates[0] == NOT_DEFINED)
        printf("Candidate set not defined\n");
    else
    {
        // Print the candidates array contents
        printf("The Candidate set C_{%d,%d} = { ", target_number, k);
        for(i=0; i < logical_size; i++)                // O(2ˆz)
            printf("%d ", candidates[i]);
        printf("}\n");
    }
}

// Finding candidate set for suffixes of original target binary number.
void candidateSetSuffix( int suffix_start, int length, int k, int z, int *Zarray, int *candidates)
{
    // Note that this entire if clause, the for loop in the last else clause
    // notwithstanding, runs in constant time
    // Also, the original target_number and its candidate set are not altered in any way
    if(suffix_start != 0)                 // if the suffix is the same as the
                                          // original number there’s no need to execute this
    {
        int max;                          // the max element in the suffix candidate set
        int i;                            // iteration variable
        int suffix_length = length - suffix_start;
        int z_suffix = z - Zarray[suffix_start-1] ;  // z_suffix  is the number of zeros
                                                      // in the suffix
        unsigned int mask =  (1 << suffix_length)-1; // this mask of the from
                                                      // 000...01111..1 (with
                                                      // suffix_length number of ’1’s)
                                                      // will be used to take the
                                                      // "suffix" of a binary number
        // perform checks
        if ((k < 1) || (k > (1 << suffix_length)))
            candidates[0] = NOT_DEFINED;
        else if (k == 1)
            max = candidates[0] & mask;  // no need to set logical_length since it was
                                         // set above -- this is because k is not variable;
                                         // also, the max candidate element is the
                                         // only element
        else if ((z_suffix == 0 && k > 1 && k <= (1 << suffix_length))
                  || (suffix_length >= z_suffix && z_suffix > 0 && k > (1 << z_suffix)))
            logical_size = 0;                               // the candidate set is empty
        else if (suffix_length >= z_suffix && z_suffix > 0 && 2 <= k && k <=(1<< z_suffix))
            max = candidates[(1 << z_suffix) - 1] & mask;  // find the max element and
                                                            // take its suffix
        if(candidates[0] == NOT_DEFINED)
            printf("Candidate set not defined\n");
        else if (logical_size == 0)
            printf("The Candidate set is empty\n");
        else
        {
            printf("The Candidate set of the suffix:\nC_{%d,%d} = { ", candidates[0] & mask, k);
            for(i = 0; i < (1<<z_suffix); i++)       // O(2ˆz_suffix)
                printf("%d ", candidates[i] & mask);
            printf("}\n");
            char *mes = "The mask and q-1 (the index of the last element) are:";
            printf("%s \n mask=%d q-1=%d \n", mes, mask , (1 << z_suffix)-1);
                                    // can output the suffix
                                    // candidate set by giving the
                                    // two defining numbers q and mask
        }
    }
}

// This function recursively computes the candidate set elements (albeit unsorted). RT: O(2ˆz)
void  generate(unsigned int i, int p)
{
    if (p == GENERATE)
    {
        aux[tail_index] = i; // keep the generated elements in the aux array
        tail_index++;
    }
    else
    {
        // the candidate set at this point holds the indices of the rightmost 0
        // to the left of given position
        int pos = candidates[p];         // find index of rightmost 0 to the left of
                                         // position p
        generate(i, pos);
        generate(i + ( 1 << (length - p - 1)), pos);      // set the 0 to a 1 and recurse
    }
}

// This function performs pre-processing using the target_number and fills the candidates array
// in order to answer the "rightmost zero" query in constant time.
// More specifically, candidates[p] is the index of the rightmost zero to the left of index p
// NON-INCLUSIVE. RT: O(length)
void create_rightmost(unsigned int target_number)
{
    int i,j,flag;
    j = GENERATE;
    flag = FALSE;                    // the flag is required because we must remember what
                                     // we saw in the previous iteration;
                                     // if flag = TRUE we saw a 0 otherwise we saw a 1
    for(i = length-1; i >= -1; i--)
    {
        if(flag)
            j = length-i-2;
        if((target_number & (1 << i)) == 0)
            flag = TRUE;
        else
            flag = FALSE;
        candidates[length-i-1] = j;
    }
}

// This function sorts the candidate set elements. RT: O(2ˆz)
void funky_sort(void)
{
    candidates[0] = 0; // the 0th index is always the first one to be visited
    int i, offset, power;
    offset = 0;
    power = 1 << z;
    // this algorithm performs a single pass over the candidates array and
    // creates each new index by manipulating appropriately  previous ones
    for(i = 1; i < (1 << z); i++)                        // O(2ˆz) time
    {
        // if i is a power of 2 reset offset and adjust power
        if((i & (i - 1))== 0)                            // even though this test accepts
                                                         // 0 as a power of 2 in this case
                                                         // this is exactly what we want!!!
        {
            power = power >> 1;
            offset = 0;
        }
        candidates[i] = candidates[offset] +  power;     // this creates the sequence of
                                                         // indices which when used to
                                                         // index in the aux array will
                                                         // sort the candidate set elements
        offset++;
    }
    for(i = 0; i < (1 << z); i++)                        // O(2ˆz) time
        candidates[i] = aux[candidates[i]];              // this is where the sorting happens
}

// Wrapper of malloc which performs a check after the system call
void *mymalloc(int size)
{
    void *p= malloc(size);
    if(p == NULL)
    {
        fprintf(stderr, "Memory allocation failed\n");
        exit(1);
    }
    return p;
}
```